# Looping

## Week 8

**MEMS 1140—Introduction to Programming in Mechanical Engineering**

# Learning Objectives (L.O.)

At the end of this lecture, you should understand/be able to:

- ❑ What `for` loops are;
- ❑ Write a `for` loop;
- ❑ Apply a `for` loop to solve a friction problem;
- ❑ What `while` loops are;
- ❑ Write a `while` loop;
- ❑ Apply a `while` loop to load a bridge model.

# Table of Contents (ToC)

# 1 – What is a `for` Loop

➩ L.O.1
❏ L.O.2
❏ L.O.3
❏ L.O.4
❏ L.O.5
❏ L.O.6

Lecture 2.2 provides the following definition of loops:

**Definition**

The program executes one or more operations repeatedly until a termination condition is met.

There are two types of loops in MATLAB

# 1 – `for` Loops



`for` loops iterate through a vector until reaching the end.

They are used in cases when the number of iterations that the loop should perform is already known.

This could arise if you have an array of known size that you want to operate on.

Lecture 2.2 gives a pack of hot dogs as an example of this.

# 2 – Writing `for` Loops



The **`for`** and **`end`** keywords are wrappers for the loop:

```
for <var> = <vector>
  <loop operations>
end
```

The variable `<var>` changes its value with each iteration of the loop according to the contents of `<vector>`.

The `<loop operations>` execute for each iteration of the loop, and often utilize `<var>` to change slightly each time.

# 2 – A Simple `for` Loop

✓ L.O.1
⇨ L.O.2
❏ L.O.3
❏ L.O.4
❏ L.O.5
❏ L.O.6

Let's inspect this with a simple example.

# 2 – A Simple `for` Loop



First set up the structure of the for loop:

```
for .
  .
end
```

# 2 – A Simple `for` Loop

✓ L.O.1
➩ L.O.2
❏ L.O.3
❏ L.O.4
❏ L.O.5
❏ L.O.6

Then declare the variable to change each iteration:

```
for looping_variable = .
  .
end
```

# 2 – A Simple `for` Loop



Then define the array of values for `looping_variable`:

```
for looping_variable = [1, 5, 3, 9, 100]
  .
end
```

# 2 – A Simple `for` Loop



Then define the operation of each loop:

```
for looping_variable = [1, 5, 3, 9, 100]
  fprintf('looping_variable = %d\n', looping_variable)
end
```

In this case, just print out the value of `looping_variable`.

# 2 – A Simple `for` Loop



## Iteration 1:

```
for looping_variable = [1, 5, 3, 9, 100]
  fprintf('looping_variable = %d\n', looping_variable)
end
```

```
looping_variable = 1
.
.
.
.
```

Command Window Output

# 2 – A Simple for Loop



## Iteration 2:

```
for looping_variable = [1, 5, 3, 9, 100]
  fprintf('looping_variable = %d\n', looping_variable)
end
```

```
looping_variable = 1
looping_variable = 5
.
.
.
```

Command Window Output

# 2 – A Simple `for` Loop



## Iteration 3:

```
for looping_variable = [1, 5, 3, 9, 100]
  fprintf('looping_variable = %d\n', looping_variable)
end
```

```
looping_variable = 1
looping_variable = 5
looping_variable = 3
.
.
```

Command Window Output

# 2 – A Simple `for` Loop



Iteration 4:

```
for looping_variable = [1, 5, 3, 9, 100]
  fprintf('looping_variable = %d\n', looping_variable)
end
```

```
looping_variable = 1
looping_variable = 5
looping_variable = 3
looping_variable = 9
.
```

Command Window Output

# 2 – A Simple `for` Loop

✓ L.O.1
➩ L.O.2
❏ L.O.3
❏ L.O.4
❏ L.O.5
❏ L.O.6

## Iteration 5:

```
for looping_variable = [1, 5, 3, 9, 100]
  fprintf('looping_variable = %d\n', looping_variable)
end
```

```
looping_variable = 1
looping_variable = 5
looping_variable = 3
looping_variable = 9
looping_variable = 100
```

Command Window Output

# 2 – Accessing an Array



Most often, `for` loops define an indexing variable in order to access different elements of arrays inside the loop.

For example:

```
x = linspace(0,5,6);
.
    .
.
```

# 2 – Accessing an Array



Most often, `for` loops define an indexing variable in order to access different elements of arrays inside the loop.

For example:

```
x = linspace(0,5,6);
for index = 1 : length(x)
   .
end
```

Each iteration, `index` takes a value from `1` to `length(x)`.

# 2 – Accessing an Array



Most often, **for** loops define an indexing variable in order to access different elements of arrays inside the loop.

For example:

```
x = linspace(0,5,6);
for index = 1 : length(x)
  fprintf('x(%d) = %.1f\n', x(index))
end
```

Using **index**, a different element of **x** is printed each iteration.

# 2 – Accessing an Array



The previous loop results in the following printout:

```
x(1) = 0.0
x(2) = 1.0
x(3) = 2.0
x(4) = 3.0
x(5) = 4.0
x(6) = 5.0
```

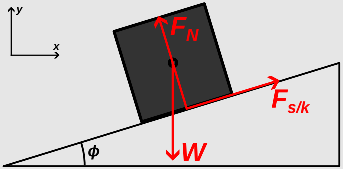
Command Window Output

Each element of the array **x** was accessed individually.

# 3 – Static/Sliding Block on an Incline



Consider the following block on an inclined plane:




$$\vec{W} = 0\hat{\imath} - W\hat{\jmath}$$

$$\vec{F}_N = \left[\left\|\vec{W}\right\|\cos(\phi)\right]\left(-\sin(\phi)\hat{\imath} + \cos(\phi)\hat{\jmath}\right)$$

$$\vec{F}_{s/k} = \left[\left\|\vec{F}_N\right\|\mu_{s/k}\right]\left(\cos(\phi)\hat{\imath} + \sin(\phi)\hat{\jmath}\right)$$

These equations allow us to simulate the net forces on the block for any inclination value $\phi$.

# 3 – Behavior of Static Friction



First, note that static friction is not always at its maximum theoretical value.

It balances other forces in magnitude until its maximum value is reached, at which point it would begin to slide.

In code, we will include a check against the static friction threshold to accurately represent the force's value.

# 3 – Preparation for the Loop



First, define the coefficients of static and kinetic friction.

```
mu_s = 0.5;    % coefficient of static friction
mu_k = 0.19;   % coefficient of kinetic friction

.
.
.
.

.
```

# 3 – Preparation for the Loop



Then, define the weight vector of the block.

```
mu_s = 0.5;    % coefficient of static friction
mu_k = 0.19;   % coefficient of kinetic friction

W = [0, -10];    % weight vector of the block
.
.
.

.
```

# 3 – Preparation for the Loop



And then functions for the normal and friction forces.

```
mu_s = 0.5;    % coefficient of static friction
mu_k = 0.19;   % coefficient of kinetic friction

W = [0, -10];    % weight vector of the block
F_N = @(phi) norm(W) * cos(phi) * [sind(phi), cosd(phi)];
.
.

.
```

# 3 – Preparation for the Loop



And then functions for the normal and friction forces.

```
mu_s = 0.5;    % coefficient of static friction
mu_k = 0.19;   % coefficient of kinetic friction

W = [0, -10];    % weight vector of the block
F_N = @(phi) norm(W) * cos(phi) * [sind(phi), cosd(phi)];
F_s = @(phi) norm(F_N) * mu_s * [-cosd(phi), sind(phi)];
.

.
```

# 3 – Preparation for the Loop



And then functions for the normal and friction forces.

```
mu_s = 0.5;    % coefficient of static friction
mu_k = 0.19;   % coefficient of kinetic friction

W = [0, -10];    % weight vector of the block
F_N = @(phi) norm(W) * cos(phi) * [sind(phi), cosd(phi)];
F_s = @(phi) norm(F_N(phi)) * mu_s * [-cosd(phi), sind(phi)];
F_k = @(phi) norm(F_N(phi)) * mu_k * [-cosd(phi), sind(phi)];
```

.

# 3 – Preparation for the Loop



Then, prepare an array of angles to evaluate.

```
mu_s = 0.5;    % coefficient of static friction
mu_k = 0.19;   % coefficient of kinetic friction

W = [0, -10];    % weight vector of the block
F_N = @(phi) norm(W) * cosd(phi) * [-sind(phi), cosd(phi)];
F_s = @(phi) norm(F_N(phi)) * mu_s * [cosd(phi), sind(phi)];
F_k = @(phi) norm(F_N(phi)) * mu_k * [cosd(phi), sind(phi)];

phi = linspace(0,90,1000);    % incline angle array to test
```

# 3 – Evaluating Forces



First set up the `for` loop to test each angle:

```
for i = 1 : length(phi)
  .
  .
  .
     .
     .
  .
     .
     .
  .
end
```

# 3 – Evaluating Forces



In the loop, solve for $\vec{F}_N$ at the given angle:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); .
   .
   .
     .
     .
   .
     .
     .
   .
end
```

# 3 – Evaluating Forces



Then solve for the maximum $\vec{F}_s$ at the given angle:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); static_friction_limit = F_s(phi(i));
   .
   .
     .
     .
   .
     .
     .
   .
end
```

# 3 – Evaluating Forces



Compare $\vec{F}_s$ to $\vec{W}$ directed down the incline:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); static_friction_limit = F_s(phi(i));
  force_ratio = abs(W(2))*sind(phi(i)) / norm(static_friction_limit);
   .
     .
     .
   .
     .
     .
   .
end
```

# 3 – Evaluating Forces



Then set up the **if** ... **else** ... statement:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); static_friction_limit = F_s(phi(i));
  force_ratio = abs(W(2))*sind(phi(i)) / norm(static_friction_limit);
  if force_ratio <= 1    % the block is not sliding yet
     .
     .
  else                   % the block is now sliding
     .
     .
  end
end
```

# 3 – Evaluating Forces



If the block is not sliding, correct the value of $\vec{F}_s$:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); static_friction_limit = F_s(phi(i));
  force_ratio = abs(W(2))*sind(phi(i)) / norm(static_friction_limit);
  if force_ratio <= 1    % the block is not sliding yet
    friction_force = force_ratio * static_friction_limit;
    .
  else                   % the block is now sliding
    .
    .
  end
end
```

# 3 – Evaluating Forces



We know the block is stationary, so set $\Sigma \vec{F} = \vec{0}$:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); static_friction_limit = F_s(phi(i));
  force_ratio = abs(W(2))*sind(phi(i)) / norm(static_friction_limit);
  if force_ratio <= 1    % the block is not sliding yet
    friction_force = force_ratio * static_friction_limit;
    sum_forces(i,:) = [0,0];
  else                   % the block is now sliding
    .
    .
  end
end
```

# 3 – Evaluating Forces



If the block is sliding, calculate the kinetic friction $\vec{F}_k$:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); static_friction_limit = F_s(phi(i));
  force_ratio = abs(W(2))*sind(phi(i)) / norm(static_friction_limit);
  if force_ratio <= 1    % the block is not sliding yet
    friction_force = force_ratio * static_friction_limit;
    sum_forces(i,:) = [0,0];
  else                   % the block is now sliding
    friction_force = F_k(phi(i));
    .
  end
end
```

# 3 – Evaluating Forces



Add the three forces to calculate $\Sigma\vec{F}$:

```
for i = 1 : length(phi)
  normal_force = F_N(phi(i)); static_friction_limit = F_s(phi(i));
  force_ratio = abs(W(2))*sind(phi(i)) / norm(static_friction_limit);
  if force_ratio <= 1    % the block is not sliding yet
    friction_force = force_ratio * static_friction_limit;
    sum_forces(i,:) = [0,0];
  else                   % the block is now sliding
    friction_force = F_k(phi(i));
    sum_forces(i,:) = W + normal_force + friction_force;
  end
end
```

# 3 – Plotting



The following code plots $\boldsymbol{\Sigma F_x}$ and $\boldsymbol{\Sigma F_y}$ against $\phi$:

```
plot(phi,sum_forces(:,1),'.--b',...
     phi,sum_forces(:,2),'.--r')
legend('$\Sigma F_{x}$','$\Sigma F_{y}$',...
       'Interpreter','latex',...
       'Location','southwest')
xlabel('$\phi$ [-]','Interpreter','latex')
ylabel('$\Sigma F$ [N]','Interpreter','latex')
set(gca, 'FontSize', 14)
```

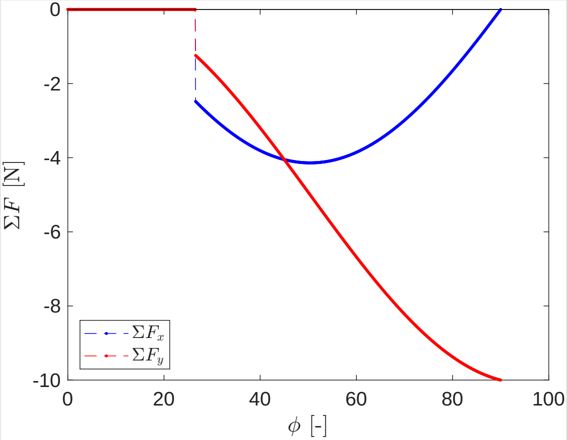
# 3 – Plotting



Note that both $\Sigma F_x$ and $\Sigma F_y$ are 0 until a sudden change.

This is the point at which the block starts to slide.

Let's figure out what that angle $\phi$ is.



# 3 – Extract the Slipping Point



First, evaluate a `logical` array of the nonzero force pairs.

```
nonzero_forces = any( not(sum_forces == [0,0]) , 2);
.
.
```

The `any` command evaluates the `or` operation along dimension `2` (each row).

This creates an array that specifies whether the net force on the block is `[0,0]` for each angle in `phi`.

# 3 – Extract the Slipping Point



The **logical** array extracts elements from **phi**.

```
nonzero_forces = any( not(sum_forces == [0,0]) , 2);
slipping_angles = phi( nonzero_forces );
.
```

Only the elements of **phi** where **nonzero_forces** evaluates to **true** are extracted.

In other words, **slipping_angles** contains only the angles where the block is slipping on the inclined plane.

# 3 – Extract the Slipping Point



Finally, we extract first element of **`slipping_angles`**.

```
nonzero_forces = any( not(sum_forces == [0,0]) , 2);
slipping_angles = phi( nonzero_forces );
slipping_point = slipping_angles(1)
```

This represents the specific angle at which the block begins to slip down the inclined plane.

```
slipping_point = 26.5766
```

Command Window Output

# 4 – `while` Loops



`while` loops iterate until the associated conditional statement no longer evaluates to `true`.

They are used when the number of iterations for the loop is not already known.

This could arise if, for example, you have a trigger variable or a threshold to meet for iteration-varying residuals.

# 5 – Writing `while` loops



The `while` and `end` keywords are wrappers for the loop:

```
while <conditional statement>
  <loop operations>
end
```

The `<conditional statement>` is evaluated at the start of each iteration of the loop and the loop continues if it is `true`.

The `<loop operations>` execute and update a variable that is considered in the `<conditional statement>`.

# 5 – A Simple `while` Loop Example



Mathworks documentation uses the factorial operation as an example:

# 5 – A Simple `while` Loop Example



First define the number on which to operate:

```
number = 10; .
.
   .
   .
.
.
```

# 5 – A Simple `while` Loop Example



Then prepare the next number for the factorial operation:

```
number = 10; next_number = 9; .
.
  .
  .
.
.
```

# 5 – A Simple `while` Loop Example



Then define a variable to update the result:

```
number = 10; next_number = 9; factorial_result = number;
.
    .
    .
.
.
```

# 5 – A Simple `while` Loop Example



Then set up the structure of the loop:

```
number = 10; next_number = 9; factorial_result = number;
while .
  .
  .
end
.
```

# 5 – A Simple while Loop Example



Then define the condition for the loop:

```
number = 10; next_number = 9; factorial_result = number;
while next_number > 1
   .
   .
end
.
```

# 5 – A Simple `while` Loop Example



Then update the result with the new number in the sequence:

```
number = 10; next_number = 9; factorial_result = number;
while next_number > 1
  factorial_result = factorial_result * next_number;
  .
end
.
```

# 5 – A Simple `while` Loop Example



Then update the next number, which is checked each iteration:

```
number = 10; next_number = 9; factorial_result = number;
while next_number > 1
  factorial_result = factorial_result * next_number;
  next_number = next_number - 1;
end
.
```

# 5 – A Simple `while` Loop Example



Finally, print the result:

```
number = 10; next_number = 9; factorial_result = number;
while next_number > 1
  factorial_result = factorial_result * next_number;
  next_number = next_number - 1;
end
fprintf('%d! = %d', number, factorial_result)
```

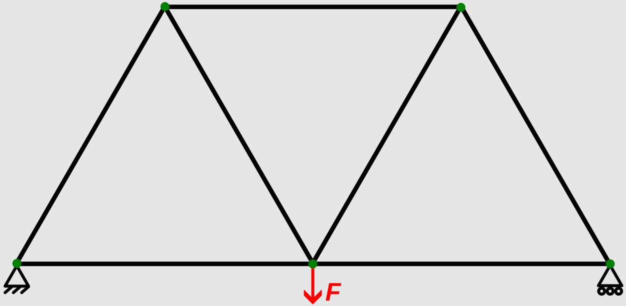
```
10! = 3628800
```

Command Window Output

# 6 – Application of `while` Loops

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4
✓ L.O.5
⇨ L.O.6

This example will use **`while`** loops to iteratively load the following truss structure until it reaches failure:



# 6 – Black-Box Evaluation Function



This demonstration will not cover how to evaluate the truss structure.

For now, we'll use some "black-box" function that encapsulates that evaluation process.

That way, we can just work with the results without needing to know how it works.

# 6 – Loading the Bridge



First set up constants of the problem:

```
yield_stress = 19.9e6;                 % [Pa]
cross_sectional_area = .1;             % [m^2]

.
.
.
    .
    .
.
```

# 6 – Loading the Bridge



Then initialize the stress vector and the applied load:

```
yield_stress = 19.9e6;                 % [Pa]
cross_sectional_area = .1;             % [m^2]

internal_stresses = zeros(1,7);        % [Pa]
force = 0;                             % [N]
.
    .
    .
.
```

# 6 – Loading the Bridge



Then set up the structure of the loop:

```
yield_stress = 19.9e6;              % [Pa]
cross_sectional_area = .1;          % [m^2]

internal_stresses = zeros(1,7);     % [Pa]
force = 0;                          % [N]
while .
  .
  .
end
```

# 6 – Loading the Bridge



Then define the condition for the loop:

```
yield_stress = 19.9e6;                   % [Pa]
cross_sectional_area = .1;               % [m^2]

internal_stresses = zeros(1,7);          % [Pa]
force = 0;                               % [N]
while max(abs(internal_stresses)) < yield_stress
   .
   .
end
```

# 6 – Loading the Bridge



Then fill in the loop operations:

```
yield_stress = 19.9e6;                  % [Pa]
cross_sectional_area = .1;              % [m^2]

internal_stresses = zeros(1,7);         % [Pa]
force = 0;                              % [N]
while max(abs(internal_stresses)) < yield_stress
  force = force + 1;
  internal_stresses = evaluate_warren(force, cross_sectional_area);
end
```

# 6 – Optimization Wrapper



The previous code determines the force at which the members in the truss exceed their yield stress.

This can be wrapped in an optimization loop that changes the truss structure to determine the *best* truss:

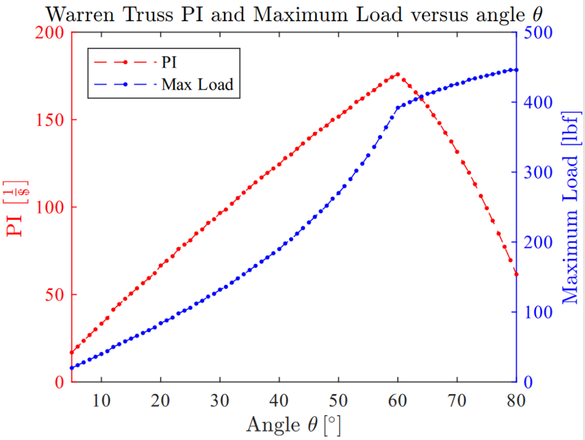
```
for angle = 1 : 90
  maximum_force(angle) = evaluate_maximum_force(angle);
  performance(angle) = calculate_performance(maximum_force, angle);
end
```

# 6 – Optimization Results



A combination of **`for`** and **`while`** loops enables a thorough analysis of the truss design.

*More complex analysis represented here.*



# 7 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4
✓ L.O.5
✓ L.O.6

This lecture covered:

- ✓ What `for` loops are

  For loops are used to iterate through a defined array and execute some operation a set number of times.

- ✓ How to write a `for` loop

  The `for` and `end` keywords surround looping operations, which execute for each value in an array defined at the start. Often, the looping variable is used as an index for accessing arrays.

# 7 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4
✓ L.O.5
✓ L.O.6

✓ Apply a `for` loop to solve a friction problem

By utilizing custom functions to evaluate applied friction and normal forces at a given angle, the `for` loop enables you to analyze the forces on a block at each angle of an inclined plane.

✓ What `while` loops are

While loops are used to iterate for an undefined number of times by using a trigger condition of some kind.

# 7 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4
✓ L.O.5
✓ L.O.6

✓ How to write a `while` loop

The `while` and `end` keywords surround looping operations, so long as the given conditional statement evaluates to true at the start of each iteration.

✓ Apply a `while` loop to load a bridge model

By using a function that evaluates the stresses with the bridge given an applied force, the while loop enables you to increment the load until the stress exceeds its limit.